AVF Control Number: AVF-IABG-060

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #891207I1.10262
TARTAN LABORATORIES INCORPORATED
Tartan Ada ULTRIX/68K Version INT-2
MicroVAX II to Tektronix 8541 Emulator for 68020

Completion of On-Site Testing:
7 December 1989

Prepared By:
IABG mbH, Abt SZT
Einsteinstr 20
D8012 Ottobrunn
West Germany

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	7 Dec. 90 ro 7 Dec. 91	Final

4. TITLE AND SUBTITLE Ada Compiler Validation Summary Report:Tartan Laboratories Incorporated, Tartan Ada ULTRIX/68K, Version INT-2(Host) MicroVAX II to Tektronix 8541 Emulator for 68020 (Target), 891207I1.10262

5. FUNDING NUMBERS

6. AUTHOR(S)
IABG-AVF
Ottobrunn, FEDERAL REPUBLIC OF GERMANY

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
LABG-AVF, Industrieanlagen-Betriebsgeselschaft
Dept. SZT
Einsteinstrasse 20
D-8012 Ottobrunn
FEDERAL REPUBLIC OF GERMANY

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF-IABG-060

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Trtan Laboratories Incorporated, Tartan Ada ULTRIX/68K Version INT-2, Ottob- -n West Germany, MicroVAX II under ULTRIX V2.2 (Host) to Tektronix 8541 Emulator for 68020 under TekDB V5.04 (Target), ACVC 1.10.

14. SUBJECT TERMS Ada programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, Validation Testing, Ada Validation Office, Ada Validation Facility, ANSI/MIL-STD-1815A, Ada Joint Program Office

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	

Ada Compiler Validation Summary Report:

Compiler Name: Tartan Ada ULTRIX/68K Version INT-2

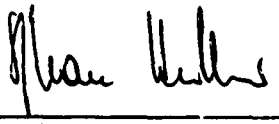
Certificate Number: #891207I1.10262

Host: MicroVAX II under ULTRIX V2.2

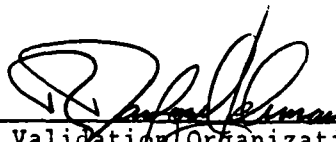
Target: Tektronix 8541 Emulator for 68020 under TekDB V5.04

Testing completed 7 December 1989 using ACVC 1.10.

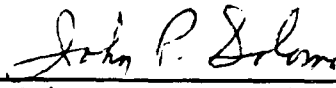
This report has been reviewed and is approved.



Dr. S. Heilbrunner
IABG mbH, Abt SZT
Einsteinstr 20
D8012 Ottobrunn
West Germany



Ada Validation Organization
for Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311



Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

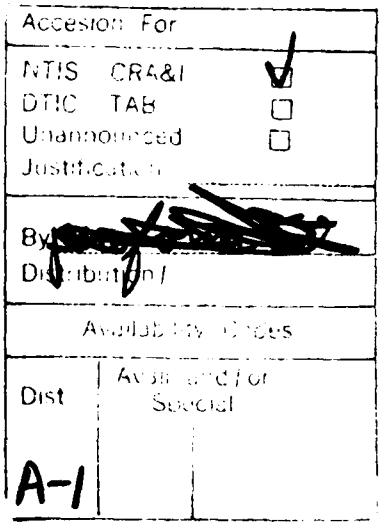


TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT 1
1.2 USE OF THIS VALIDATION SUMMARY REPORT 2
1.3 REFERENCES 3
1.4 DEFINITION OF TERMS 3
1.5 ACVC TEST CLASSES 4

CHAPTER 2 CONFIGURATION INFORMATION 7

2.1 CONFIGURATION TESTED 7
2.2 IMPLEMENTATION CHARACTERISTICS 8

CHAPTER 3 TEST INFORMATION 13

3.1 TEST RESULTS 13
3.2 SUMMARY OF TEST RESULTS BY CLASS 13
3.3 SUMMARY OF TEST RESULTS BY CHAPTER 14
3.4 WITHDRAWN TESTS 14
3.5 INAPPLICABLE TESTS 14
3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS . 18
3.7 ADDITIONAL TESTING INFORMATION
3.7.1 Prevalidation 19
3.7.2 Test Method 19
3.7.3 Test Site 20

APPENDIX A DECLARATION OF CONFORMANCE

APPENDIX B APPENDIX F OF THE Ada STANDARD

APPENDIX C TEST PARAMETERS

APPENDIX D WITHDRAWN TESTS

APPENDIX E COMPILER AND LINKER OPTIONS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent, but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by IABG mbH, Abt SZT according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO).

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

IABG mbH, Abt SZT
Einsteinstr 20
D8012 Ottobrunn

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.

Inapplicable test — An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.

Passed test — An ACVC test for which a compiler generates the expected result.

Target — The computer which executes the code generated by the compiler.

Test — A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.

Withdrawn test — An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation and execution of legal Ada programs with certain language constructs which cannot be verified at run time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and tests. However, some tests contain values that require the test to be

customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: Tartan Ada ULTRIX/68K Version INT-2

ACVC Version: 1.10

Certificate Number: #891207I1.10262

Host Computer:

Machine:	MicroVAX II
Operating System:	ULTRIX V2.2
Memory Size:	9 MB

Target Computer:

Machine:	Tektronix 8541 Emulator for 68020
Operating System:	TekDB V5.04
Memory Size:	64 KB + 1 MB

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)

4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

1) This implementation supports the additional predefined types SHORT_INTEGER, BYTE_INTEGER, and LONG_FLOAT in the package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

1) None of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

3) This implementation uses no extra bits for extra precision and uses all extra bits for extra range. (See test C35903A.)

4) NUMERIC_ERROR is raised for predefined and largest integer and no exception is raised for smallest integer when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

5) No exception is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

6) Underflow is gradual. (See tests C45524A..Z (26 tests).)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

1) The method used for rounding to integer is round to even. (See tests C46012A..Z (26 tests).)

2) The method used for rounding to longest integer is round to even. (See tests C46012A..Z (26 tests).)

3) The method used for rounding to integer in static universal real expressions is round to even. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR for one dimensional array types, two dimensional array types and two dimensional array subtypes, and no exception for one dimensional array subtypes. (See test C36003A.)

2) NUMERIC_ERROR is raised when an array type with INTEGER'LAST + 2 components is declared. (See test C36202A.)

3) NUMERIC_ERROR is raised when an array type with SYSTEM.MAX_INT + 2 components is declared. (See test C36202B.)

4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array type is declared. (See test C52103X.)

5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array type is declared and exceeds INTEGER'LAST. (See test C52104Y.)

6) In assigning one-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

7) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

8) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

f. Discriminated types.

1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

1) In the evaluation of a multi-dimensional aggregate, the test results indicate that all choices are evaluated before checking against the index type. (See tests C43207A and C43207B.)

2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

3) CONSTRAINT_ERROR is raised after all choices are

evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

1) The pragma INLINE is supported for functions and procedures. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

i. Generics.

This compiler enforces the following two rules concerning declarations and proper bodies which are individual compilation units:

o generic bodies must be compiled and completed before their instantiation.

o recompilation of a generic body or any of its transitive subunits makes all units obsolete which instantiate that generic body.

These rules are enforced whether the compilation units are in separate compilation files or not. AI408 and AI506 allow this behaviour.

1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

2) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

3) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)

4) Generic non-library package bodies as subunits can be compiled in separate compilations. (See test CA2009C.)

5) Generic non-library subprogram bodies can be compiled in separate compilations from their stubs. (See test CA2009F.)

6) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

7) Generic package declarations and bodies can be

compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

8) Generic library package specifications and bodies can be compiled in separate compilations. (See tests BC3204C and BC3205D.)

9) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

j. Input and output.

1) The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

2) The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

3) The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits this behavior for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO.

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 512 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation, and for 238 executable tests that use file operations not supported by the implementation. Modifications to the code, processing, or grading for 81 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	129	1132	1821	17	16	46	3161
Inapplicable	0	6	494	0	12	0	512
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	198	577	545	245	172	99	162	332	133	36	252	334	76	3161
N/A	14	72	135	3	0	0	4	0	4	0	0	35	245	512
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

E28005C	A39005G	B97102E	C97116A	BC3009B	CD2A62D
CD2A63A	CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B
CD2A66C	CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D
CD2A76A	CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G
CD2A84N	CD2A84M	CD50110	CD2B15C	CD7205C	CD2D11B
CD5007B	ED7004B	ED7005C	ED7005D	ED7006C	ED7006D
CD7105A	CD7203B	CD7204B	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B				

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 519 tests were inapplicable for the reasons indicated:

a. The following 201 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests) C35705L..Y (14 tests)
C35706L..Y (14 tests) C35707L..Y (14 tests)

C35708L..Y (14 tests) C35802L..Z (15 tests)
C45241L..Y (14 tests) C45321L..Y (14 tests)
C45421L..Y (14 tests) C45521L..Z (15 tests)
C45524L..Z (15 tests) C45621L..Z (15 tests)
C45641L..Y (14 tests) C46012L..Z (15 tests)

b. C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.

c. The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:

C45231C C45304C C45502C C45503C C45504C
C45504F C45611C C45613C C45614C C45631C
C45632C B52004D C55B07A B55B09C B86001W
CD7101F

d. C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because the value of SYSTEM.MAX_MANTISSA is less than 32.

e. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

f. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

g. CA2009A, CA2009C, CA2009F and CA2009D are not applicable because this compiler creates dependencies between generic bodies, and units that instantiate them (see section 2.2i for rules and restrictions concerning generics).

h. CD1009C, CD2A41A..E (5 tests), and CD2A42A..J (10 tests) are not applicable because this implementation imposes restrictions on 'SIZE length clauses for floating point types.

i. CD2A61I is not applicable because this implementation imposes restrictions on 'SIZE length clauses for array types.

j. CD2A84B..I (8 tests) and CD2A84K..L (2 tests) are not applicable because this implementation imposes restrictions on 'SIZE length clauses for access types.

k. CD2A91A..E (5 tests) are not applicable because 'SIZE length clauses for task types are not supported.

l. CD2B11G is not applicable because 'STORAGE_SIZE representation clauses are not supported for access types where the designated type is a task type.

m. CD2B15B is not applicable because a collection size larger than the size specified was allocated.

n. The following 238 tests are inapplicable because sequential, text, and direct access files are not supported:

CE2102A..C (3 tests)	CE2102G..H (2 tests)
CE2102K	CE2102N..Y (12 tests)
CE2103C..D (2 tests)	CE2104A..D (4 tests)
CE2105A..B (2 tests)	CE2106A..B (2 tests)
CE2107A..H (8 tests)	CE2107L
CE2108A..B (2 tests)	CE2108C..H (6 tests)
CE2109A..C (3 tests)	CE2110A..D (4 tests)
CE2111A..I (9 tests)	CE2115A..B (2 tests)
CE2201A..C (3 tests)	CE2201F..N (9 tests)
CE2204A..D (4 tests)	CE2205A
CE2208B	CE2401A..C (3 tests)
CE2401E..F (2 tests)	CE2401H..L (5 tests)
CE2404A..B (2 tests)	CE2405B
CE2406A	CE2407A..B (2 tests)
CE2408A..B (2 tests)	CE2409A..B (2 tests)
CE2410A..B (2 tests)	CE2411A
CE3102A..B (2 tests)	EE3102C
CE3102F..H (3 tests)	CE3102J..K (2 tests)
CE3103A	CE3104A..C (3 tests)
CE3107B	CE3108A..B (2 tests)
CE3109A	CE3110A
CE3111A..B (2 tests)	CE3111D..E (2 tests)
CE3112A..D (4 tests)	CE3114A..B (2 tests)
CE3115A	EE3203A
CE3208A	EE3301B
CE3302A	CE3305A
CE3402A	EE3402B
CE3402C..D (2 tests)	CE3403A..C (3 tests)
CE3403E..F (2 tests)	CE3404B..D (3 tests)
CE3405A	EE3405B
CE3405C..D (2 tests)	CE3406A..D (4 tests)
CE3407A..C (3 tests)	CE3408A..C (3 tests)
CE3409A	CE3409C..E (3 tests)
EE3409F	CE3410A
CE3410C..E (3 tests)	EE3410F
CE3411A	CE3411C
CE3412A	EE3412C
CE3413A	CE3413C
CE3602A..D (4 tests)	CE3603A
CE3604A..B (2 tests)	CE3605A..E (5 tests)
CE3606A..B (2 tests)	CE3704A..F (6 tests)
CE3704M..O (3 tests)	CE3706D
CE3706F..G (2 tests)	CE3804A..P (16 tests)
CE3805A..B (2 tests)	CE3806A..B (2 tests)
CE3806D..E (2 tests)	CE3806G..H (2 tests)

CE3905A..C (3 tests) CE3905L
CE3906A..C (3 tests) CE3906E..F (2 tests)

These tests were not processed because their inapplicability can be deduced from the result of other tests.

o. Tests CE2103A..B (2 tests) and CE3107A raise USE_ERROR although NAME_ERROR is expected. These tests report FAILED but they were graded not applicable because this implementation does not support permanent files.

p. EE2201D, EE2201E, EE2401D, EE2401G are inapplicable because sequential, text, and direct access files are not supported.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that was not anticipated by the test (such as raising one exception instead of another).

Modifications were required for 81 tests.

a. The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B22003A	B24007A	B24009A	B25002B	B32201A	B34005N
B34005T	B34007H	B35701A	B36171A	B36201A	B37101A
B37102A	B37201A	B37202A	B37203A	B37302A	B38003A
B38003B	B38008A	B38008B	B38009A	B38009B	B38103A
B38103B	B38103C	B38103D	B38103E	B43202C	B44002A
B48002A	B48002B	B48002D	B48002E	B48002G	B48003E
B49003A	B49005A	B49006A	B49007A	B49009A	B4A010C
B54A20A	B54A25A	B58002A	B58002B	B59001A	B59001C
B59001I	B62006C	B67001A	B67001B	B67001C	B67001D
B74103E	B74104A	B85007C	B91005A	B95003A	B95007B
B95031A	B95074E	BC1002A	BC1109A	BC1109C	BC1206A
BC2001E	BC3005B	BC3009C	BD5005B		

b. For the two tests BC3204C and BC3205D, the compilation order was changed to

BC3204C0, C1, C2, C3M, C4, C5, C6, C3M

and

BC3205D0, D2, D1M

respectively. This change was necessary because of the compiler's rules for separately compiled generic units (see section 2.2i for rules and restrictions concerning generics). When processed in this order the expected error messages were produced for BC3204C3M and BC3205D1M.

c. The two tests BC3204D and BC3205C consist of several compilation units each. The compilation units for the main procedures are near the beginning of the files. When processing these files unchanged, a link error is reported instead of the expected compiled generic units. Therefore, the compilation files were modified by appending copies of the main procedures to the end of

these files. When processed, the expected error messages were generated by the compiler.

d. Tests C39005A, CD7004C, CD7005E and CD7006E wrongly presume an order of elaboration of the library unit bodies. These tests were modified to include a PRAGMA ELABORATE (REPORT);

e. Test E28002B checks that predefined or unrecognized pragmas may have arguments involving overloaded identifiers without enough contextual information to resolve the overloading. It also checks the correct processing of pragma LIST. For this implementation, pragma LIST is only recognised if the compilation file is compiled without errors or warnings. Hence, the test was modified to demonstrate the correct processing of pragma LIST.

f. Tests C45524A and C45524B contain a check at line 136 that may legitimately fail as repeated division may produce a quotient that lies within the smallest safe interval. This check was modified to include, after line 138, the text:

```
ELSIF VAL <= F'SAFE_SMALL THEN COMMENT ("UNDERFLOW SEEMS GRADUAL");
```

For this implementation, the required support package specification, SPPRT13SP, was rewritten to provide constant values for the function names.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the Tartan Ada ULTRIX/68K Version INT-2 compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the Tartan Ada ULTRIX/68K Version INT-2 compiler using ACVC Version 1.10 was conducted by IABG on the premises of TARTAN. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host Computer:

Machine:	MicroVAX II
Operating System:	ULTRIX V2.2
Memory Size:	9 MB

Target Computer:

Machine: Tektronix 8541 Emulator for 68020
Operating System: TekDB V5.04
Memory Size: 64 KB + 1 MB

Compiler:

Tartan Ada ULTRIX/68K Version INT-2

The original ACVC was customized prior to the validation visit in order to remove all withdrawn tests, inapplicable I/O tests and tests requiring unsupported floating point precisions. Tests that make use of implementation specific values were also customized. Tests requiring modifications during the prevalidation testing were modified accordingly.

A tape containing the customized ACVC was read by the host computer.

After the test files were loaded to disk, the full set of tests was compiled and linked. All executable tests were transferred via an RS232 line to the target computer where they were run. Results were transferred to the host computer in the same way, where they were evaluated and archived.

The compiler was tested using command scripts provided by TARTAN LABORATORIES INCORPORATED and reviewed by the validation team. The compiler was tested using no option settings. All chapter B tests were compiled with the listing option on (i.e. -La). The linker was called with the command

alib68 link <testname>

A full list of compiler and linker options is given in Appendix E.

3.7.3 Test Site

Testing was conducted at TARTAN LABORATORIES INCORPORATED, Pittsburgh and was completed on 7 December 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

TARTAN LABORATORIES INCORPORATED has submitted the following Declaration of Conformance concerning the Tartan Ada ULTRIX/68K Version INT-2 compiler.

DECLARATION OF CONFORMANCE

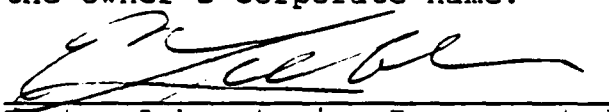
Compiler Implementor: Tartan Laboratories Incorporated
Ada Validation Facility: IABG mbH, Dept. SZT
Ada Compiler Validation Capability (ACVC) Version: 1.10

Base Configuration

Base Compiler Name:	Tartan Ada ULTRIX/68K
Base Compiler Version:	Version INT-2
Host Computer:	Micro VAX II under ULTRIX V2.2
Target Computer:	Tektronix 8541 Emulator for 68020

Implementor's Declaration

I, the undersigned, representing Tartan Laboratories Incorporated, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that Tartan Laboratories Incorporated is the owner of record of the Ada Language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada Language compiler(s) listed in this declaration shall be made only in the owner's corporate name.

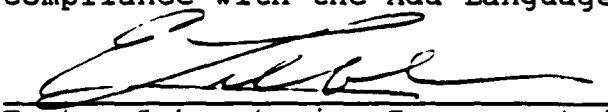


Date: Dec. 7, 1989

Tartan Laboratories Incorporated
Ed Lieblein, Sr. VP, Development

Owner's Declaration

I, the undersigned, representing Tartan Laboratories Incorporated, take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I declare that all of the Ada Language compilers listed, and their host/target performance, are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.



Date: Dec. 7, 1989

Tartan Laboratories Incorporated
Ed Lieblein, Sr. VP, Development

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the Tartan Ada ULTRIX/68K Version INT-2 compiler, as described in this Appendix, are provided by TARTAN LABORATORIES INCORPORATED. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, are contained in Appendix F.

Chapter 5
Appendix F to MIL-STD-1815A

This chapter contains the required Appendix F to *Military Standard, Ada Programming Language*, ANSI/MIL-STD-1815A (American National Standards Institute, Inc., February 17, 1983).

5.1. PRAGMAS

5.1.1. Predefined Pragmas

This section summarizes the effects of and restrictions on predefined pragmas.

- Access collections are not subject to automatic storage reclamation so pragma CONTROLLED has no effect. Space deallocated by means of UNCHECKED_DEALLOCATION will be reused by the allocation of new objects.
- Pragma ELABORATE is supported.
- Pragma INLINE is supported. The body for an inlined subprogram need not appear in the same compilation unit as the call. Inlining will take place only if the subprogram body is present in the library and is not obsolete.
- Pragma INTERFACE is not supported. The implementation-defined pragma FOREIGN_BODY (see Section 5.1.2.2) can be used to interface to subprograms written in other languages.
- Pragma LIST is supported but has the intended effect only if the command line option -La was supplied for compilation, and the listing generated was not due to the presence of errors and/or warnings.
- Pragma OPTIMIZE is supported except when at the outer level (that is, in a package specification or body).
- Pragma PACK is fully supported.
- Pragma PAGE is supported but has the intended effect only if the command line option -La was supplied for compilation, and the listing generated was not due to the presence of errors and/or warnings.
- Pragma PRIORITY is fully supported.
- Pragma SUPPRESS is fully supported as required by Ada LRM 11.7.
- Future releases of the compiler will support the following pragmas: MEMORY_SIZE, SHARED, STORAGE_UNIT and SYSTEM_NAME.

A warning message will be issued if an unsupported pragma is supplied.

5.1.2. Implementation-Defined Pragmas

Implementation-defined pragmas provided by Tartan are described in the following sections.

5.1.2.1. Pragma LINKAGE_NAME

The pragma LINKAGE_NAME associates an Ada entity with a string that is meaningful externally; e.g., to a linkage editor. It takes the form

pragma LINKAGE_NAME (*Ada-simple-name, string-constant*)

The *Ada-simple-name* must be the name of an Ada entity declared in a package specification. This entity must be one that has a runtime representation: e.g., a subprogram, exception or object. It may not be a named number or string constant. The pragma must appear after the declaration of the entity in the same package specification.

The effect of the pragma is to cause the *string-constant* to be used in the generated assembly code as an external name for the associated Ada entity. It is the responsibility of the user to guarantee that this string constant is meaningful to the linkage editor and that no illegal linkname clashes arise.

5.1.2.2. Pragma FOREIGN_BODY

A subprogram written in another language can be called from an Ada program. Pragma FOREIGN_BODY is used to indicate that the body for a non-generic top-level package specification is provided in the form of an object module. The bodies for several subprograms may be contained in one object module.

Use of the pragma FOREIGN_BODY dictates that all subprograms, exceptions and objects in the package are provided by means of a foreign object module. In order to successfully link a program including a foreign body, the object module for that body must be provided to the library using the a11b68 foreign command described in Section 4.7.

The pragma is of the form:

pragma FOREIGN_BODY (*language_name* [, *elaboration_routine_name*])

The parameter *language_name* is a string intended to allow the compiler to identify the calling convention used by the foreign module (but this functionality is not yet in operation). Currently, the programmer must ensure that the calling convention and data representation of the foreign body procedures are compatible with those used by the Tartan Ada compiler. Subprograms called by tasks should be reentrant.

The optional *elaboration_routine_name* string argument provides a means to initialize the package. The routine specified as the *elaboration_routine_name*, which will be called for the elaboration of this package body, must be a global routine in the object module provided by the user.

A specification that uses this pragma may contain only subprogram declarations, object decarations that use an unconstrained type mark, and number declarations. Pragmas may also appear in the package. The type mark for an object cannot be a task type, and the object declaration must not have an initial value expression. The pragma must be given prior to any declarations within the package specification. If the pragma is not located before the first declaration, or any restriction on the declarations is violated, the pragma is ignored and a warning is generated.

The foreign body is entirely responsible for initializing objects declared in a package utilizing pragma FOREIGN_BODY. In particular, the user should be aware that the implicit initializations described in LRM 3.2.1 are not done by the compiler. (These implicit initializations are associated with objects of access types, certain record types and composite types containing components of the preceding kinds of types.)

Pragma LINKAGE_NAME should be used for all declarations in the package, including any declarations in a nested package specification to be sure that there are no conflicting link names. If pragma LINKAGE_NAME is not used, the cross-reference qualifier, -x, (see Section 3.2) should be used when invoking the compiler and the resulting cross-reference table of linknames inspected to identify the linknames assigned by the compiler and determine that there are no conflicting linknames (see also Section 3.5).

In the following example, we want to call a function p1mn which computes polynomials and is written in C.

```
package MATH_FUNCS is
  pragma FOREIGN_BODY ("C");
  function POLYNOMIAL  (X:INTEGER) return INTEGER;
        --Ada spec matching the C routine
  pragma LINKAGE_NAME (POLYNOMIAL, "plmn");
        --Force compiler to use name "plmn" when referring to this
  -- function
end MATH_FUNCS;

with MATH_FUNCS; use MATH_FUNCS
procedure MAIN is
  X:INTEGER := POLYNOMIAL(10);
        -- Will generate a call to "plmn"
  begin ...
end MAIN;
```

To compile, link and run the above program, you do the following steps:

1. Compile `MATH_FUNCS`
2. Compile `MAIN`
3. Obtain an object module (e.g. `math.toff`) containing the compiled code for `plmn`, converted to TOFF; if the module is written in assembly code, for example, using the `basys_to_toff` utility (See *Object File Utilities*, Chapter 4)
4. Issue the command

   ```
   alib68 foreign math_funcs math.toff
   ```

5. Issue the command

   ```
   alib68 link main
   ```

Without Step 4, an attempt to link will produce an error message informing you of a missing package body for `MATH_FUNCS`.

Using an Ada body from another Ada program library. The user may compile a body written in Ada for a specification into the library, regardless of the language specified in the pragma contained in the specification. This capability is useful for rapid prototyping, where an Ada package may serve to provide a simulated response for the functionality that a foreign body may eventually produce. It also allows the user to replace a foreign body with an Ada body without recompiling the specification.

The user can either compile an Ada body into the library, or use the command `alib68 foreign` (See Section 4.7) to use an Ada body from another library. The Ada body from another library must have been compiled under an identical specification. The pragma `LINKAGE_NAME` must have been applied to all entities declared in the specification. The only way to specify the linkname for the elaboration routine of an Ada body is with the pragma `FOREIGN_BODY`.

Using Calls to the Operating System. In some cases, the foreign code is actually supplied by the operating system (in the case of system calls) or by runtime libraries for other programming languages such as C. Such calls may be made using a dummy procedure to supply a file specification to the `alib68 foreign`command. You need a dummy `.toff` file which may be obtained in a number of ways. One way is to compile the procedure

```
procedure DUMMY is
begin
   null;
end;
```

Then, use the library command

```
alib68 foreign pkg dummy.toff
```

where *pkg* is the name of the package that contains the pragma `LINKAGE_NAME` for the operating system call.

For example to use the ULTRIX system call _sbrk in the program TEST:

```
Package MEMORY is
   pragma FOREIGN_BODY ("ASM");
   procedure GET_VIRTUAL_MEMORY(MEM:INTEGER);
   pragma LINKAGE_NAME (GET_VIRTUAL_MEMORY, "_sbrk ");
end MEMORY;

with MEMORY;
procedure TEST is
      ...

begin
      GET_VIRTUAL_MEMORY(MEM);
     ...
end TEST;
```

Obtain the file dummy.toff. Then use

```
alib68 foreign memory dummy.toff
```

to include the body for the system call in the library.

5.2. IMPLEMENTATION-DEPENDENT ATTRIBUTES

No implementation-dependent attributes are currently supported.

5.3. SPECIFICATION OF THE PACKAGE SYSTEM

The parameter values specified for the 68K target in package SYSTEM [LRM 13.7.1 and Annex C] are:

```
package SYSTEM is
   type ADDRESS is new INTEGER;
   type NAME is (MC68000);
   SYSTEM_NAME   : constant NAME := MC68000;
   STORAGE_UNIT  : constant := 8;
   MEMORY_SIZE   : constant := 1_000_000;
   MAX_INT       : constant := 2_147_483_647;
   MIN_INT       : constant := -MAX_INT - 1;
   MAX_DIGITS    : constant := 15;

   MAX_MANTISSA  : constant := 31;
   FINE_DELTA    : constant := 2#1.0#e-31;
   TICK          : constant := 0.01;
   subtype PRIORITY is INTEGER range 10 .. 200;
   DEFAULT_PRIORITY : constant PRIORITY := PRIORITY'FIRST;
   RUNTIME_ERROR    : exception;
end SYSTEM;
```

5.4. RESTRICTIONS ON REPRESENTATION CLAUSES

The following sections explain the basic restrictions for representation specifications followed by additional restrictions applying to specific kinds of clauses.

5.4.1. Basic Restriction

The basic restriction on representation specifications [LRM 13.1] that they may be given only for types declared in terms of a type definition, excluding a generic_type_definition (LRM 12.1) and a private_type_definition (LRM 7.4). Any representation clause in violation of these rules is not obeyed by the compiler; a diagnostic message is issued.

Further restrictions are explained in the following sections. Any representation clauses violating those restrictions are not obeyed but cause a diagnostic message to be issued.

5.4.2. Length Clauses

Length clauses [LRM 13.2] are, in general, supported. For details, refer to the following sections.

5.4.2.1. Size Specifications for Types

The rules and restrictions for size specifications applied to types of various classes are described below.

The following principle rules apply:

1. The size is specified in bits and must be given by a static expression.
2. The specified size is taken as a mandate to store objects of the type in the given size wherever feasible. No attempt is made to store values of the type in a smaller size, even if possible. The following rules apply with regard to feasibility:

   - An object that is not a component of a composite object is allocated with a size and alignment that is referable on the target machine; that is, no attempt is made to create objects of non-referable size on the stack. If such stack compression is desired, it can be achieved by the user by combining multiple stack variables in a composite object; for example

```
type My_Enum is (A,B);
for My_enum'size use 1;
V,W: My_enum; -- will occupy two storage
              -- units on the stack
              -- (if allocated at all)
type rec is record
   V,W: My_enum;
end record;
pragma Pack(rec);
O: rec; -- will occupy one storage unit
```

   - A formal parameter of the type is sized according to calling conventions rather than size specifications of the type. Appropriate size conversions upon parameter passing take place automatically and are transparent to the user.

   - Adjacent bits to an object that is a component of a composite object, but whose size is non-referable, may be affected by assignments to the object, unless these bits are occupied by other components of the composite object; that is, whenever possible, a component of non-referable size is made referable.

   In all cases, the compiler generates correct code for all operations on objects of the type, even if they are stored with differing representational size in different contexts.

   Note: A size specification cannot be used to force a certain size in value operations of the type; for example

```
type my_int is range 0..65535;
for my_int'size use 16; -- o.k.
A,B: my_int;
   ...A + B... -- this operation will generally be
               -- executed on 32-bit values
```

3. A size specification for a type specifies the size for objects of this type and of all its subtypes. For components of composite types, whose subtype would allow a shorter representation of the component, no attempt is made to take advantage of such shorter representations. In contrast, for types without a length clause, such components may be represented in a lesser number of bits than the number of bits required to represent all values of the type. Thus, in the example

```
type MY_INT is range 0..2**15-1;
for MY_INT'SIZE use 16; -- (1)
subtype SMALL_MY_INT is MY_INT range 0..255;
type R is record
            ...
            X: SMALL_MY_INT;
            ...
          end record;
```

the component R.X will occupy 16 bits. In the absence of the length clause at (1), R.X may be represented in 8 bits.

For the following type classes, the size specification must coincide with the default size chosen by the compiler for the type:

- access types
- floating-point types
- task types

No useful effect can be achieved by using size specifications for these types.

5.4.2.2. Size Specification for Scalar Types

The specified size must accommodate all possible values of the type including the value 0 (even if 0 is not in the range of the values of the type). For numeric types with negative values the number of bits must account for the sign bit. No skewing of the representation is attempted. Thus

```
type my_int is range 100..101;
```

requires at least 7 bits, although it has only two values, while

```
type my_int is range -101..-100;
```

requires 8 bits to account for the sign bit.

A size specification for a real type does not affect the accuracy of operations on the type. Such influence should be exerted via the `accuracy_definition` of the type (LRM 3.5.7, 3.5.9).

A size specification for a scalar type may not specify a size larger than the largest operation size supported by the target architecture for the respective class of values of the type.

5.4.2.3. Size Specification for Array Types

A size specification for an array type must be large enough to accommodate all components of the array under the densest packing strategy explained below in adherence to any alignment constraints on the component type (see Section 5.4.7).

The size of the component type cannot be influenced by a length clause for an array. Within the limits of representing all possible values of the component subtype (but not necessarily of its type), the representation of components may, however, be reduced to the minimum number of bits, unless the component type carries a size specification.

If there is a size specification for the component type, but not for the array type, the component size is rounded up to a referable size, unless pragma PACK is given. This applies even to boolean types or other types that require only a single bit for the representation of all values.

5.4.2.4. Size Specification for Record Types

A size specification for a record type does not influence the default type mapping of a record type. The size must be at least as large as the number of bits determined by type mapping. Influence over packing of components can be exerted by means of (partial) record representation clauses or by Pragma PACK.

Neither the size of component types, nor the representation of component subtypes can be influenced by a length clause for a record.

The only implementation-dependent components allocated by Tartan Ada in records contain dope information for arrays whose bounds depend on discriminants of the record or contain relative offsets of components within a record layout for record components of dynamic size. These implementation-dependent components cannot be named or sized by the user.

A size specification cannot be applied to a record type with components of dynamically determined size.

Note: Size specifications for records can be used only to widen the representation accomplished by padding at the beginning or end of the record. Any narrowing of the representation over default type mapping must be accomplished by representation clauses or pragma PACK.

5.4.2.5. Specification of Collection Sizes

The specification of a collection size causes the collection to be allocated with the specified size. It is expressed in storage units and need not be static; refer to package SYSTEM for the meaning of storage units.

Any attempt to allocate more objects than the collection can hold causes a STORAGE_ERROR exception to be raised. Dynamically sized records or arrays may carry hidden administrative storage requirements that must be accounted for as part of the collection size. Moreover, alignment constraints on the type of the allocated objects may make it impossible to use all memory locations of the allocated collection. Furthermore, some administrative overhead for the allocator must be taken into account by the user (currently 1 word per allocated object).

In the absence of a specification of a collection size, the collection is extended automatically if more objects are allocated than possible in the collection originally allocated with the compiler-established default size. In this case, STORAGE_ERROR is raised only when the available target memory is exhausted. If a collection size of zero is specified, no access collection is allocated.

5.4.2.6. Specification of Task Activation Size

The specification of a task activation size causes the task activation to be allocated with the specified size. It is expressed in storage units; refer to package SYSTEM for the meaning of storage units.

Any attempt to exceed the activation size during execution causes a STORAGE_ERROR exception to be raised. Unlike collections, there is generally no extension of task activations.

5.4.2.7. Specification of 'SMALL

Only powers of 2 are allowed for 'SMALL.

The length of the representation may be affected by this specification. If a size specification is also given for the type, the size specification takes precedence; the specification of 'SMALL must then be accommodatable within the specified size.

5.4.3. Enumeration Representation Clauses

For enumeration representation clauses [LRM 13.3], the following restrictions apply:

- The internal codes specified for the literals of the enumeration type may be any integer value between INTEGER'FIRST and INTEGER'LAST. It is strongly advised to not provide a representation clause that merely duplicates the default mapping of enumeration types, which assigns consecutive numbers in ascending order starting with 0, since unnecessary runtime cost is incurred by such duplication. It should be noted that the use of attributes on enumeration types with user-specified encodings is costly at run time.
- Array types, whose index type is an enumeration type with non-contiguous value encodings, consist of a contiguous sequence of components. Indexing into the array involves a runtime translation of the index value into the corresponding position value of the enumeration type.

5.4.4. Record Representation Clauses

The alignment clause of record representation clauses [LRM 13.4] is observed. The specified expression must yield a target-dependent value.

Static objects may be aligned at powers of 2 up to a page boundary. The specified alignment becomes the minimum alignment of the record type, unless the minimum alignment of the record forced by the component allocation and the minimum alignment requirements of the components is already more stringent than the specified alignment.

The component clauses of record representation clauses are allowed only for components and discriminants of statically determinable size. Not all components need to be present. Component clauses for components of variant parts are allowed only if the size of the record type is statically determinable for every variant.

The size specified for each component must be sufficient to allocate all possible values of the component subtype (but not necessarily the component type). The location specified must be compatible with any alignment constraints of the component type; an alignment constraint on a component type may cause an implicit alignment constraint on the record type itself.

If some, but not all, discriminants and components of a record type are described by a component clause, then the discriminants and components without component clauses are allocated after those with component clauses; no attempt is made to utilize gaps left by the user-provided allocation.

5.4.5. Address clauses

Address clauses [LRM 13.5] are supported with the following restrictions:

- When applied to an object, an address clause becomes a linker directive to allocate the object at the given address. For any object not declared immediately within a top-level library package, the address clause is meaningless. Address clauses applied to local packages are not supported by Tartan Ada. Address clauses applied to library packages are prohibited by the syntax; therefore, an address clause can be applied only to a package if it is a body stub.
- Address clauses applied to subprograms and tasks are implemented according to the LRM rules. When applied to an entry, the specified value identifies an interrupt in a manner customary for the target. Immediately after a task is created, a runtime call is made for each of its entries having an address clause, establishing the proper binding between the entry and the interrupt.
- Specified addresses must be constants.

5.4.6. Pragma PACK

Pragma PACK [LRM 13.1] is supported. For details, refer to the following sections.

5.4.6.1. Pragma PACK for Arrays

If pragma PACK is applied to an array, the densest possible representation is chosen. For details of packing, refer to the explanation of size specifications for arrays (Section 5.4.2.3).

If, in addition, a length clause is applied to

1. the array type, the pragma has no effect, since such a length clause already uniquely determines the array packing method.
2. the component type, the array is packed densely, observing the component's length clause. Note that the component length clause may have the effect of preventing the compiler from packing as densely as would be the default if pragma PACK is applied where there was no length clause given for the component type.

5.4.6.2. The Predefined Type String

Package STANDARD applies Pragma PACK to the type `string`. However, when applied to character arrays, this pragma cannot be used to achieve denser packing than is the default for the target: 1 character per 8-bit word.

5.4.6.3. Pragma PACK *for Records*

If pragma PACK is applied to a record, the densest possible representation is chosen that is compatible with the sizes and alignment constraints of the individual component types. Pragma PACK has an effect only if the sizes of some component types are specified explicitly by size specifications and are of non-referable nature. In the absence of pragma PACK, such components generally consume a referable amount of space.

It should be noted that default type mapping for records maps components of boolean or other types that require only a single bit to a single bit in the record layout if there are multiple such components in a record. Otherwise, it allocates a referable amount of storage to the component.

If pragma PACK is applied to a record for which a record representation clause has been given detailing the allocation of some but not all components, the pragma PACK affects only the components whose allocation has not been detailed. Moreover, the strategy of not utilizing gaps between explicitly allocated components still applies.

5.4.7. Minimal Alignment for Types

Certain alignment properties of values of certain types are enforced by the type mapping rules. Any representation specification that cannot be satisfied within these constraints is not obeyed by the compiler and is appropriately diagnosed.

Alignment constraints are caused by properties of the target architecture, most notably by the capability to extract non-aligned component values from composite values in a reasonably efficient manner. Typically, restrictions exist that make extraction of values that cross certain address boundaries very expensive, especially in contexts involving array indexing. Permitting data layouts that require such complicated extractions may impact code quality on a broader scale than merely in the local context of such extractions.

Instead of describing the precise algorithm of establishing the minimal alignment of types, we provide the general rule that is being enforced by the alignment rules:

- No object of scalar type including components or subcomponents of a composite type, may span a target-dependent address boundary that would mandate an extraction of the object's value to be performed by two or more extractions.

5.5. IMPLEMENTATION-GENERATED COMPONENTS IN RECORDS

The only implementation-dependent components allocated by Tartan Ada in records contain dope information for arrays whose bounds depend on discriminants of the record. These components cannot be named by the user.

5.6. INTERPRETATION OF EXPRESSIONS APPEARING IN ADDRESS CLAUSES

Section 13.5.1 of the Ada Language Reference Manual describes a syntax for associating interrupts with task entries. Tartan Ada implements the address clause

```
for TOENTRY use at intID;
```

by associating the interrupt specified by `intID` with the `toentry` entry of the task containing this address clause. The interpretation of `intID` is both machine and compiler dependent.

The Motorola 680x0 specification provides 256 interrupts that may be associated with task entries. These interrupts are identified by an integer in the range 0..255, corresponding to the interrupt vector numbers in section 6.2.1 of the 68020 User's Manual. When you specify an interrupt address clause, the `intID` argument is interpreted as follows:

- If the argument is in the range 0..255, a full support interrupt association is made between the interrupt specified by the argument and the task entry. That is, the runtimes make no assumptions about the task in question. This is the slower method.
- If the argument is in the range 256..511, a fast interrupt association is made between the interrupt number (argument-256) and the task entry. This method provides faster execution because the runtimes can depend upon the assumptions previously described.

For the difference between full support and fast interrupt handling, refer to Section 8.5.

5.7. *RESTRICTIONS ON UNCHECKED CONVERSIONS*

Tartan supports `UNCHECKED_CONVERSION` with a restriction that requires the sizes of both source and target types to be known at compile time. The sizes need not be the same. If the value in the source is wider than that in the target, the source value will be truncated. If narrower, it will be zero-extended. Calls on instantiations of `UNCHECKED_CONVERSION` are made inline automatically.

5.8. *IMPLEMENTATION-DEPENDENT ASPECTS OF INPUT-OUTPUT PACKAGES*

Tartan Ada supplies the predefined input/output packages `DIRECT_IO`, `SEQUENTIAL_IO`, `TEXT_IO`, and `LOW_LEVEL_IO` as required by LRM Chapter 14. However, since the target computer is used in embedded applications lacking both standard I/O devices and file systems, the functionality of `DIRECT_IO`, `SEQUENTIAL_IO`, and `TEXT_IO` is limited.

`DIRECT_IO` and `SEQUENTIAL_IO` raise `USE_ERROR` if a file open or file access is attempted. `TEXT_IO` is supported to `CURRENT_OUTPUT` and from `CURRENT_INPUT`. A routine that takes explicit file names raises `USE_ERROR`.

5.9. *OTHER IMPLEMENTATION CHARACTERISTICS*

The following information is supplied in addition to that required by Appendix F to MIL-STD-1815A.

5.9.1. *Definition of a Main Program*

Any Ada library subprogram unit may be designated the main program for purposes of linking (using the `a11b68` command) provided that the subprogram has no parameters.

Tasks initiated in imported library units follow the same rules for termination as other tasks [described in LRM 9.4 (6-10)]. Specifically, these tasks are not terminated simply because the main program has terminated. Terminate alternatives in selective wait statements in library tasks are therefore strongly recommended.

5.9.2. *Implementation of Generic Units*

All instantiations of generic units, except the predefined generic `UNCHECKED_CONVERSION` and `UNCHECKED_DEALLOCATION` subprograms, are implemented by code duplications. No attempt at sharing code by multiple instantiations is made in this release of Tartan Ada. (Code sharing will be implemented in a later release.)

Tartan Ada enforces the restriction that the body of a generic unit must be compiled before the unit can be instantiated. It does not impose the restriction that the specification and body of a generic unit must be provided as part of the same compilation. A recompilation of the body of a generic unit will obsolete any units that instantiated this generic unit.

5.9.3. Implementation-Defined Characteristics in Package STANDARD

The implementation-dependent characteristics in package STANDARD [Annex C] are:

```
package STANDARD is
...
type BYTE_INTEGER is range -128 .. 127;
type SHORT_INTEGER is range -32768 .. 32767;
type INTEGER is range -2_147_483_648 .. 2_147_483_647;
type FLOAT is digits 6 range -16#0.7FFF_FF8#E+32 .. 16#0.7FFF_FF8#E+32;

type LONG_FLOAT is digits 9 range -16#0.7FFF_FFFF_FFFF_FE#E+256 ..
      16#0.7FFF_FFFF_FFFF_FE0#E+256 ;
type DURATION is delta 0.0001 range -86400.0 .. 86400.0;
      -- DURATION'SMALL = 2#1.0#E-14 (that is, 6.103516E-5 sec)
...
end STANDARD;
```

5.9.4. Attributes of Type Duration

The type DURATION is defined with the following characteristics:

```
DURATION'DELTA is 0.0001 sec
DURATION'SMALL is 6.103516E-5 sec
DURATION'FIRST is -86400.0 sec
DURATION'LAST is 86400.0 sec
```

5.9.5. Values of Integer Attributes

Tartan Ada supports the predefined integer types INTEGER, SHORT_INTEGER and BYTE_INTEGER. The range bounds of the predefined type INTEGER are:

```
INTEGER'FIRST = -2**31
INTEGER'LAST = 2**31-1

SHORT_INTEGER'FIRST = -2**15
SHORT_INTEGER'LAST = 2**15-1

BYTE_INTEGER'FIRST = -128
BYTE_INTEGER'LAST = 127
```

The range bounds for subtypes declared in package TEXT_IO are:

```
COUNT'FIRST = 0
COUNT'LAST = INTEGER'LAST - 1

POSITIVE_COUNT'FIRST = 1
POSITIVE_COUNT'LAST = INTEGER'LAST - 1

FIELD'FIRST = 0
FIELD'LAST = 20
```

The range bounds for subtypes declared in packages DIRECT_IO are:

```
COUNT'FIRST = 0
COUNT'LAST = INTEGER'LAST

POSITIVE_COUNT'FIRST = 1
POSITIVE_COUNT'LAST = COUNT'LAST
```

5.9.6. Values of Floating-Point Attributes

Tartan Ada supports the predefined floating-point types FLOAT and LONG_FLOAT.

Attribute	Value for FLOAT
DIGITS	6
MANTISSA	21
EMAX	84
EPSILON approximately	16#0.1000_000#E-4 9.53674E-07
SMALL approximately	16#0.8000_000#E-21 2.58494E-26
LARGE approximately	16#0.FFFF_F80#E+21 1.93428E+25
SAFE_EMAX	126
SAFE_SMALL approximately	16#0.2000_000#E-31 5.87747E-39
SAFE_LARGE approximately	16#0.3FFF_FE0#E+32 8.50706E+37
FIRST approximately	-16#0.7FFF_FFC#E+32 -1.70141E+38
LAST approximately	16#0.7FFF_FFC#E+32 1.70141E+38
MACHINE_RADIX	2
MACHINE_MANTISSA	24
MACHINE_EMAX	126
MACHINE_EMIN	-126
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE

Attribute	Value for LONG_FLOAT
DIGITS	15
MANTISSA	53
EMAX	204
EPSILON approximately	16#0.4000_0000_0000_000#E-12 8.8817841970013E-16
SMALL approximately	16#0.8000_0000_0000_000#E-51 1.94469227433 16E-62
LARGE approximately	16#0.FFFF_FFFF_FFFF_E00#E+51 2.5711008708143E+61
SAFE_EMAX	1022
SAFE_SMALL approximately	16#0.2000_0000_0000_000#E-255 1.1125369292536-308
SAFE_LARGE approximately	16#0.3FFF_FFFF_FFFF_F80#E+256 4.4942328371557E+307
FIRST approximately	-16#0.7FFF_FFFF_FFFF_FE#E+256 -8.988465674312E+307
LAST approximately	16#0.7FFF_FFFF_FFFF_FE0#E+256 8.9884656743115E+307
MACHINE_RADIX	2
MACHINE_MANTISSA	51
MACHINE_EMAX	1022
MACHINE_EMIN	-1022
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below. The use of the '*' operator signifies a multiplication of the following character, and the use of the '&' character signifies concatenation of the preceeding and following strings. The values within single or double quotation marks are to highlight character or string values:

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	239 * 'A' & '1'
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	239 * 'A' & '2'
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except for a character near the middle.	120 * 'A' & '3' & 119 * 'A'

Name and Meaning	Value
$BIG_ID4 An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.	120 * 'A' & '4' & 119 * 'A'
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	237 * '0' & "298"
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	235 * '0' & "690.0"
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	'"' & 120 * 'A' & '"'
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	'"' & 119 * 'A' & '1' & '"'
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	220 * ' '
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483646
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	1_000_000
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	MC68000
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	20
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	THERE_IS_NO_SUCH_FIXED_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	THERE_IS_NO_SUCH_FLOAT_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100_000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	100_000_000.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	200
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	/NON_EXISTENT_DIRECTORY1/FILE1
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	/NON_EXISTENT_DIRECTORY2/FILE2

Name and Meaning	Value
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-100_000_000.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	10
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	240
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648

Name and Meaning	Value
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"2:" & 235 * '0' & "11:"
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"16:" & 233 * '0' & "F.E:"
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	'"' & 238 * 'A' & '"'
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	BYTE_INTEGER
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	MC68000
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	8#777777777776#

Name and Meaning	Value
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	1_000_000
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	MC68000
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	96
$TICK A real literal whose value is SYSTEM.TICK.	0.01

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

a. E28005C This test expects that the string "-- TOP OF PAGE. -- 63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this that must appear at the top of the page.

b. A39005G This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

c. B97102E This test contains an unitended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

d. C97116A This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING_OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.

e. BC3009B This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

f. CD2A62D This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

g. CD2A63A..D, CD2A66A..D, CD2A73A..D, CD2A76A..D [16 tests] These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

h. CD2A81G, CD2A83G, CD2A84N & M, & CD50110 [5 tests] These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86 & 96, 86 & 96, and 58, resp.).

i. CD2B15C & CD7205C These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

j. CD2D11B This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

k. CD5007B This test wrongly expects an implicitly declared subprogram to be at the the address that is specified for an unrelated subprogram (line 303).

l. ED7004B, ED7005C & D, ED7006C & D [5 tests] These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

m. CD7105A This test requires that successive calls to CALENDAR.-CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).

n. CD7203B, & CD7204B These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

o. CD7205D This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

p. CE2107I This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid. (line 90)

q. CE3111C This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

r. CE3301A This test contains several calls to END_OF_LINE & END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, & 136).

s. CE3411B This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

APPENDIX E

COMPILER AND LINKER OPTIONS

Chapter 3
Compiling Ada Programs

The `tada68` command is used to compile and assemble Ada compilation units.

3.1. *THE* `tada68` *COMMAND FORMAT*

The `tada68` command has this format:

`tada68` [*option...*] *file...* [*option...*]

Arguments that start with a hyphen are interpreted as options; otherwise, they represent filenames. There must be at least one filename, but there need not be any options. Options and filenames may appear in any order, and all options apply to all filenames. For an explanation of the available options, see Section 3.2.

If a source file does not reside in the directory in which the compilation takes place, *file* must include a path sufficient to locate the file. It is recommended that only one compilation unit be placed in a file.

Files are processed in the order in which they appear on the command line. The compiler sequentially processes all compilation units in each file. Upon successful compilation of a unit:

- the Ada program library `ada.db` is updated to reflect the new compilation time and any new dependencies
- one or more separate compilation files and/or object files are generated

If no errors are detected in a compilation unit, `tada68` produces an object module and updates the library. If any error is detected, no object code file is produced, a source listing is produced, and no library entry is made for that compilation unit. If warnings are generated, both an object code file and a source listing are produced. For further details about the process of updating the library, files generated, replacement of existing files, and possible error conditions, see Sections 3.3 through 3.6.

The output from `tada68` is a file of type `.stoff` or `.toff`, for a specification or a body unit respectively, containing object code. Some other files are left in the directory as well. See Section 3.4 for a list of extensions of files that may be left in the directory.

3.2. *OPTIONS*

Command line options indicate special actions to be performed by the compiler or special output file properties.

The following command line options may be used:

`-a`	Generate an assembly code file. The assembly code file has an extension `.s` or `.ss` (see Section 3.4).
`-g`	Output debugging information.
`-i`	Cause compiler to omit data segments with the text of enumeration literals. This text is normally produced for exported enumeration types in order to support the `'IMAGE` attribute. You should use `-i` only when you can guarantee that no unit that will import the enumeration type will use `'IMAGE`. However, if you are compiling a unit with an enumeration type that is not visible to other compilation units, this option is not needed. The compiler can recognize when `'IMAGE` is not used and will not generate the supporting strings.

`-La` Generate a listing, even if no errors were found. The default is to generate a listing only if an error is found.

`-Ln` Never generate a listing. The default is to generate a listing only if an error is found.

`-n=`*n* Stop compilation after *n* errors have been detected.

`-Op`*n* Control the level of optimization performed by the compiler, requested by *n*. The optimization levels available are:

- *n* = 0 **Minimum** - Performs context determination, constant folding, algebraic manipulation, and short circuit analysis.
- *n* = 1 **Low** - Performs level 0 optimizations plus common subexpression elimination and equivalence propagation within basic blocks. It also optimizes evaluation order.
- *n* = 2 **Space** - *This is the default if none is supplied.* Performs level 1 optimizations plus flow analysis which is used for common subexpression elimination and equivalence propagation across basic blocks. It also performs invariant expression hoisting, dead code elimination, and assignment killing. Level 2 also performs lifetime analysis which is used to improve register allocation. It also performs inline expansion of subprogram calls indicated by Pragma `INLINE` which appears in the same compilation unit.
- *n* = 3 **Time** - Performs level 2 optimizations plus inline expansion of subprogram calls which the optimizer decides are profitable to expand (from an execution time perspective). Other optimizations which improve execution time at a cost to image size are performed only at this level.

`-v` Print out compiler phase names. The compiler prints out a short description of each compilation phase in progress.

`-q` Do not print out compiler phase names.

`-w` Suppress warning messages.

`-S[ACDEILORSZ]` Suppress the given set of checks:

A	ACCESS_CHECK
C	CONSTRAINT_CHECK
D	DISCRIMINANT_CHECK
E	ELABORATION_CHECK
I	INDEX_CHECK
L	LENGTH_CHECK
O	OVERFLOW_CHECK
R	RANGE_CHECK
S	STORAGE_CHECK
Z	"ZERO"DIVISION_CHECK

The `-S` option has the same effect as a global pragma `SUPPRESS` applied to the source file. If the source program also contains a pragma `SUPPRESS`, then a given check is suppressed if either the pragma or the switch specifies it; that is, the effect of a pragma `SUPPRESS` cannot be negated with the command line option. See LRM 11.7 for further details. Examples are:

- `-SOZ` Suppress `OVERFLOW_CHECK` and `"ZERO"DIVISION_CHECK`.
- `-S` Suppress all checks.
- `-SC` Suppress `CONSTRAINT_ERROR`, equivalent to `-SADILR`.

`-x` Cause the compiler to generate a cross reference file containing entries of the form

Ada-name=>linkname at line

This option will allow users to find the linkname generated for the given *Ada-name*, and use linkname to set breakpoints in debuggers. The file will have the extension `.xrf` (See Section 3.5).

In addition, the output from the compiler may be redirected using the redirection facility including '`&`' for `stderr`; for example

```
% tada68 tax_spec.ada >& tax_spec.txt
```

3.3. *WHAT UPDATES ARE MADE TO THE PROGRAM LIBRARY*

Simply stated, upon successful compilation of a unit,

- the Ada program library `ada.db` is updated to reflect the new compilation time and any new dependencies
- one or more separate compilation files and/or object files are generated.

However, more complicated situations can arise. The following items list the types of compilation unit and address the range of situations that can arise.

- In all cases the *transitive closure* of the dependencies of a compilation unit in the library must be consistent; that is, the unit must be compiled consistently as defined in section 10.3 of the LRM. A secondary unit can have its specification in its context clause, although it is redundant. For a more complete discussion of closure, see Section 4.5.
- A **package specification** replaces any library unit in the library with the same name, or is simply added if no such library unit exists.
- A **package body** replaces any existing body of a package specification with the same name. If no such specification exists, an error message is issued. If such a specification exists, but the body does not match the specification in the sense of Section 7.1 of the LRM, error messages are issued.
- A **subprogram specification** replaces any library unit in the library with the same name, or is simply added if no such library unit exists.
- A **subprogram body** replaces any existing body of a (generic or non-generic) subprogram specification with the same name. If no such specification exists, an implicit specification is derived from the body and entered into the library as noted above for subprogram specification. If a specification exists, but the body does not match the specification in the sense of Section 6.3 of the LRM, error messages are issued. If any library unit other than a subprogram specification exists with the same name, the new implicit specification replaces that library unit.
- **Generic package specifications and subprogram specifications** act as explicit specifications, i.e., corresponding bodies must match their specifications. If a generic unit is instantiated, a dependency is created on the generic body.
- **Generic instances** compiled as library units are treated in the same way as their non-generic counterparts.
- When an instantiation replaces the body of a library unit, all secondary units of that library unit are now obsolete and are deleted.
- A **subunit** with a parent unit containing an appropriate body stub existing in the library replaces any subunit with the same subunit name, comprised of the stub's name and the name of the ancestor unit, or is simply added, if no such subunit exists. A unit containing stubs will only be entered into the library if the simple names of all its stubs are unique for all stubs derived from its common ancestor. An error message is issued if no parent unit exists in the library, the parent unit exists but does not have a relevent stub, or the parent unit and body stub exist but the subunit does not match the stub or its specification.

- When the parent unit of a **subunit** is recompiled and the parent no longer contains a stub for the subunit, the subunit, which is now obsolete, is deleted.

3.4. *FILES PRODUCED OR USED BY THE COMPILATION SYSTEM*

Files with the following extensions are contained in the standard packages directories or can be created by compiling or linking an Ada program; the file name is the name of a compilation unit, but may be compressed to conform to length limitations.

`bod`	Representation of the body of a generic, and/or the visibility information available to any subunits or generic bodies. *body-name*`.bod` is read when compiling a program that instantiates *body-name* or a generic contained in a *body-name*, or is a subunit of *body-name*.
`di`	Representation of a unit specification. *unit-name*`.di` is read during the compilation of a program that does a "**with** *unit-name*".
`lst`	A listing produced by the Ada compiler.
`map`	A link file produced by the Tartan linker.
`s`	The assembly language file produced by compiling an Ada unit body when the `-a` option is given.
`stoff`	The object file produced by compiling an Ada unit specification.
`toff`	The object file produced by compiling an Ada unit body.
`xrf`	Cross reference files that relate Ada names with compressed and disambiguated names used in the object or assembly language file.
`xtof`	An executable image created by linking a main program.

The following are the extensions used for files that are created temporarily during the linking process:

`etof`	The elaboration script generated by the library
`lls`	List of all object files to be linked by the linker
`lcf`	Control file for the linker.

Additionally, temporary files are created during compilation that have the same file extensions listed above, but also have a unique 8 digit hexadecimal number concatenated to the extension. Any of the above files will appear in the directory only if a link or compilation is abnormally terminated. These files should then be deleted by the user.

The parser creates several additional temporary files having the extensions `ter`, `teb`, `tsb`, and `tse` that should be deleted by the user if left by an interrupted compilation.

Files having the following extensions are controlled by the librarian and compiler: `di`, `bod`, `toff` and `stoff`. The user should not use these extensions for any other purpose. `alib68 delete` command will automatically delete these files, when the respective unit is deleted from the library. If the user deletes these files in any other way, subsequent invocations of the compiler or librarian will have unpredictable results, including fatal crashes. We therefore advise that the user *never* delete these files by operating system commands.

3.5. *THE CROSS REFERENCE MAP OF LINKNAMES*

A cross reference of symbolic names to linknames is generated with the option `-x` to the `tada68` command. The cross-reference file has the extension `.xrf`: the file name is that of the compiled unit, but possibly compressed to match restrictions, as shown in the example below.

For longer unit-names, the cross reference file generated may not have an immediately obvious name, in order to find it, it may be necessary to search the current working directory for a recently produced file with extention `xrf`.

Example:

File crexample_spec.ada

```
package THIS_IS_A_LONG_PACKAGE_NAME is
    package ANOTHER_LONG_PACKAGE_NAME is
      procedure LONG_PROCEDURE_NAME_THAT_WILL_HAVE_SHORT_LINKNAME;
    end ANOTHER_LONG_PACKAGE_NAME;
end THIS_IS_A_LONG_PACKAGE_NAME;
```

File crexample_body.ada

```
package body THIS_IS_A_LONG_PACKAGE_NAME is
    package body ANOTHER_LONG_PACKAGE_NAME is separate;
end THIS_IS_A_LONG_PACKAGE_NAME;
```

File crexample.sep

```
separate (THIS_IS_A_LONG_PACKAGE_NAME)
package body ANOTHER_LONG_PACKAGE_NAME is
  procedure LONG_PROCEDURE_NAME_THAT_WILL_HAVE_SHORT_LINKNAME is
    begin
      null;
    end LONG_PROCEDURE_NAME_THAT_WILL_HAVE_SHORT_LINKNAME;
end ANOTHER_LONG_PACKAGE_NAME;
```

The commands:

```
tada68 crexample_spec.ada
tada68 crexample_body.ada
tada68 -x crexample.sep
```

produce the file thsslngpckgnm.nthrlngpckgnme001.xrf which appears below:

```
----- Linkname Cross Reference Map thsslngpckgnm.nthrlngpckgnme001 -----

this_is_a_long_package_name=>xxthsskgnm001 at       0
this_is_a_long_package_nameYthis_is_a_long_package_name=>this_is_a_long
_package_nameY00 at       1
this_is_a_long_package_nameYanother_long_package_nameYanother_long_package
_name=>another_long_package_nameY00 at       2
this_is_a_long_package_nameYanother_long_package_nameYlong_procedure_name
_that_will_have_short_linkname=>xxthsskgnm001YlngprodtLnknmY00 at       3
```

In the above cross reference file:

- The first line represents the name for the elaboration code for the package THIS_IS_A_LONG_PACKAGE_NAME. The symbols representing the specification and body have respectively YDECLARE and YBODY postpended.
- The second line is the name of the data segment for the package THIS_IS_A_LONG_PACKAGE_NAME.
- The third line is the name of the data segment for the package ANOTHER_LONG_PACKAGE_NAME.
- The name for the LONG_PROCEDURE_NAME_THAT_WILL_HAVE_SHORT_LINKNAME procedure is on the fourth line.
- The fifth line is the name for the elaboration variable for the procedure LONG_PROCEDURE_NAME_THAT_WILL_HAVE_SHORT_LINKNAME. Ada rules require that the body of a subprogram is already elaborated before it can be called. If it is not already elaborated the exception PROGRAM_ERROR must be raised. For each subprogram that may require an elaboration check the compiler generates a variable that is used to record that the body of the subprogram has been elaborated. The name of the elaboration variable is generated by postpending YGOTO to the name of the subprogram. The elaboration variable name is then subject to the same compression algorithms as the rest of the symbols in the program.

The ULTRIX command

```
ls -lt *.xrf
```

will help you locate the cross reference listing. In order to view the contents of the cross reference file, make sure that your terminal is set to wrap around mode. The identifier appearing at the left is the identifier that appears in the Ada source code. The name to the right of the '=>' is the linkname that is supplied for that identifier to `ld`. The "`at` *<number>*" gives the line number in the source code where the identifier is found. A Cross Reference Map can be used to verify that there are no conflicting linknames in a program library that uses subprograms written in another language (see Section 5.1.2.2 that discusses the pragma `FOREIGN_BODY`). It is also useful for assembly-level debugging.

3.6. *COMPILER DIAGNOSTIC MESSAGES*

The compiler and library issue *diagnostic messages* that appear at your terminal and in the optional compiler-generated listing. Most messages issued by Tartan Ada ULTRIX/68K contain a reference to the Ada LRM section and paragraph relevant to the error. This section explains the kinds of diagnostic messages that are generated, how the compiler attempts to deal with problems that caused the messages and how you should go about correcting a program.

A comprehensive listing of all the messages the compiler can issue is contained in Appendix Section A.1. A similar listing of all the messages the library can issue is contained in Appendix Section A.2.

3.6.1. *Message Severity Levels*

Every message issued by the compiler is assigned a *severity level* that indicates how serious the problem is. There are four message categories.

1. A *fatal error* is serious enough to suspend compilation immediately after the error is discovered. This is the only class of error that inhibits further analysis of the source program. An example of a fatal error message (in this case from the library) is:

```
Fatal 6801: <library administration file name> is incompatible
  with this version of the library.
```

2. An *error* is serious enough to prevent the generation of object code, but the compiler attempts to recover from the error and continues checking the source for additional errors. An example of an error is:

```
Error 2060: This record field has already been assigned
  in the aggregate (4.3 (6))
```

3. A *warning* does not stop the compiler from generating object code, but may still be an indication of a programming error. When a warning occurs, the code generated may not be what you intended. An example of a warning is:

```
Warn 4001: Elaboration of this subtype will raise constraint_error
  at runtime (3.3.2).
```

4. An *informational* message provides you with additional information when you use some library commands (see, for example, Section 4.6). Informational messages are issued only by the library, not by the compiler. An example of an informational message is:

```
Info 6011: The files required for linking by <unit_kind>
  <unit_name> are consistent (10.3).
```

3.6.2. *Message Formats*

The format of messages appearing on the standard error output and in the listing file is similar. Here is an example:

```
   5|         s1 : string(1 .. discrim);
   6|         s2 : string(1 .. 2 * discrim);
                                   ^1
***  1 Error 2204: A discriminant may not be used in this
***      expression (3.7.1 (6))
   7|    end record;
```

The numbered lines in the example are lines from the source program. The source line in question is followed by the messages and pointers to the exact location of the problem.

On the terminal, horizontal lines are used to separate messages coming from different parts of the source program, for example:

```
   5|         s1 : string(1 .. discrim);
   6|         s2 : string(1 .. 2 * discrim);
                                   ^1
***  1 Error 2204: A discriminant may not be used
***      in this expression (3.7.1 (6))
   7|    end record;
```

```
  14|       null;
  15|    when numeric_error | constraint_error =>
                              ^1
***  1 Error 3112: A given exception may only appear once
***      in a handler (11.2 (5))
  16|       null;
```

In a listing file, a message chain accompanies each diagnostic message. The message chain indicates where in the program the next and previous messages occur, for example:

```
Ada ULTRIX/68K INT-2  Copyright 1989, Tartan Laboratories Incorporated

*** First diagnostic is on line 6

   1|procedure sample_program is
   2|   subtype small_int is integer range 1 .. 10;
   3|
   4|   type rec(discrim : small_int) is record
   5|        s1 : string(1 .. discrim);
   6|        s2 : string(1 .. 2 * discrim);
                                  ^1
***  1 Error 2204: A discriminant may not be used in this
***     expression  (3.7.1 (6))
*** Next diagnostic is on line 15
   7|   end record;
   8|
   9|   x : rec(5);
  10|begin
  11|   x := (6, "12345", "abcde");
  12|exception
  13|   when constraint_error =>
  14|      null;
  15|   when numeric_error | constraint_error =>
                            ^1
*** Previous diagnostic was on line 6
***  1 Error 3112: A given exception may only appear once
***     in a handler (11.2 (5))
  16|      null;
  17|end sample_program;
  18|

*** Last diagnostic was on line 15

*** Errors: 2, Warnings: 0
```

The message chain is especially helpful when working with large listings.

Whether on the standard error output or in a listing file, the list of messages is followed by a summary line containing a count of the number of errors in each severity class, for example:

```
? Errors: 2, Warnings: 1
```

3.6.3. Message Generation

Tartan Ada ULTRIX/68K has many internal phases, any one of which can issue diagnostic messages. Messages are collected in memory until the time comes to generate the message listing. At that time, all the messages are sorted by their position in the source program and are printed.

When you examine a program listing containing many messages, remember that the order in which the messages appear in the listing is not necessarily the order in which the messages were generated. This fact may be important when one error causes another. It is advisable to start correcting your program according to the messages having the lowest numbers and work towards the higher numbers, making an intermediate compilation if necessary.

3.6.4. About Syntax Errors and Recovery

Tartan Ada ULTRIX/68K incorporates a parser which is capable of analyzing and correcting all syntactic errors in the source program. This section describes the various error messages that may be issued by the parser. When a syntax error is detected, no object code is generated.

The parser divides the source program text into lexical elements, or *tokens*, such as identifiers, reserved words, constants, etc. When the parser encounters a token that it does not expect, it issues an error message that

indicates the position at which the error was detected and the action that was taken to correct the error. Here are some examples of the recovery actions:

In the example below, the trailing "$" does not match any of the valid tokens of Ada and so the parser deletes it.

```
    1|procedure bad_syntax is
    2|   subtype byte is integer range 0 .. 255; $
                                                 ^1
***  1 Error 104: Ill-formed token deleted.
    3|   x : integer;
```

The compiler also deletes a token occupying an inappropriate place. In the following example, it deletes the superfluous token "while".

```
    5|begin
    6|   for while i in 1 .. 10 loop
             ^1
***  1 Error 121: Parse error; token deleted.
    7|      x := x + 1;
```

In the following example, the missing symbol ";" is inserted, and parsing continues undisturbed.

```
    4|   i : integer
    5|   b : boolean;
         ^1
***  1 Error 120: Parse error; token ";" inserted.
    6|begin
```

In the following example, the syntactically incorrect symbol ":" is replaced by the proper symbol ";".

```
   17|   end loop;
   18|   x := 15:
                ^1
***  1 Error 122: Parse error; this token deleted, ";" inserted.
   19|
```

3.6.4.1. Multi-Token Insertion

The parser may also insert several tokens in an attempt to repair the constructs whose closing tokens (e.g., "end if;") are missing. An example of this recovery is:

```
   10|       end if;
   11|   end loop;
         ^1,2,3
***  1 Error 120: Parse error; token "if" inserted.
***  2 Error 120: Parse error; token ";" inserted.
***  3 Error 120: Parse error; token "end" inserted.
   12|end bad_proc;
```

In this example an additional "end if;" was missing. Note, however, that the maximum number of tokens that can be inserted in succession is limited.

3.6.4.2. Complex Recovery Strategy

If all the simple fix-up and multi-token insertion techniques above are unsuccessful, the parser attempts a more massive correction by deleting many successive or preceding phrases of the program. For example,

```
    5|   y := 6;     case bad_case is
                     ****************
    6*                 when x=y => x := 3;
                       ^1
***  1 Error 123: Parse error; ill-formed "<statement>".
    7*                 when 2 := x = 5;
```

The caret (^) locates the place where the parser detects an error. The compiler indicates the elided portion of the source program by underlining with asterisks. In the first (and possibly the last line), only the tokens deleted

are underlined. The lines following the first line are not underlined, but when the entire line is deleted, the symbol after the line number (e.g., lines 6 and 7) changes from a vertical bar (|) to an asterisk. (The above case statement contained three errors. The closing "`end case;`" was also deleted.)

In cases like the above example, when the parser has deleted more than one token, the error message is

```
Error 123: Parse error; ill-formed "<name>".
```

The name contained within the angle bracket pair is that of the grammatical element that the parser expected to find in this position. Occasionally, the deletion of tokens starts at a point textually preceding the reported error because no legal interpretation of an enclosing construct can be found due to the error within the construct.

Under rare circumstances, you may see an error message

```
Error 127: Parse error; unexpected end-of-file.
```

pointing to a token within the program, with the rest of the program marked as deleted. This message points to the position in the program where the parser detected a syntax error. It indicates that, despite all attempts, the error recovery was unsucessful until the end-of-file was reached. In this case, correct all the errors reported and examine the program for missing keywords that end complicated syntactic constructs: also, especially examine the few lines that precede and follow this message for syntactic errors.

4.8. *THE* `link` *SUBCOMMAND*

The `link` command checks that the unit within the library specified by the user has the legal form for a main unit, checks that all its dependencies are consistent, finds all required object files, and links the main program with its full closure (See Section 4.5) producing an executable image. The format of the `link` command is

```
alib68 link [option...] Ada-name [option...]
```

where the *Ada-name* specifies the unit in the library to be made the main program.

The following options may be used:

`-t` Provide a trace of the load command indicating what files are being loaded

`-K` Create a shell script file that may be redirected to `sh` to cause the Ada program to be linked. The user assumes full responsibility for the consistency of the program when this script is run instead of using the `alib68 link` command.

`-M` Provide a load map

`-o` *filename*
Name the final output file from the loader *filename*